Heuristique statique améliorée d'ordonnancement de tâches : impact sur le tri des tâches et sur l'allocation de processeur

PENGCHENG MU [1], JEAN-GABRIEL COUSIN [1], JEAN-FRANÇOIS NEZAN [1], MICKAËL RAULET [1]

[1] IETR / groupe Image et Télédétection
UMR CNRS 6164 / INSA de Rennes
20 avenue des Buttes de Coësmes, CS 14315, 35043 Rennes Cedex, France

[1] pmu@insa-rennes.fr, jcousin@insa-rennes.fr, jnezan@insa-rennes.fr, mraulet@insa-rennes.fr

Résumé – L'ordonnancement de tâches est une étape importante dans le prototypage rapide d'applications de traitement d'images sur des systèmes parallèles embarqués. Nous présentons ainsi dans cet article une heuristique statique améliorée d'ordonnancement par liste : d'une part, cette heuristique intègre de nouvelles règles de priorité de tâches, tenant compte de la contention sur les communications entre tâches ; d'autre part, cette heuristique affine l'allocation d'un processeur à une tâche courante, en impactant le choix du processeur par un ordonnancement partiel de la tâche successeur critique (« *critical child* ») à la tâche courante. Nos résultats expérimentaux soulignent une accélération effective de l'application implantée, dans un contexte de moyenne comme de forte communication.

Abstract – Task scheduling is an important step in a rapid prototyping of digital image processing applications on parallel embedded systems. This paper presents advanced techniques for a static list scheduling heuristic with communication contention. First, new node priorities are used to generate static node lists. Second, the critical child technique selects a suitable processor for a node by taking into account not only its scheduled predecessors, but also its most important unscheduled successor (critical child). The experimental results show that our method is effective to reduce the schedule length in all the cases of medium and high communication.

1 Introduction

Les systèmes embarqués récents sont constitués de plusieurs processeurs de traitement du signal (DSP) ou se présentent sous forme de systèmes sur puce MPSoC (« MultiProcessor System on Chip »), promettant ainsi traitements parallèles et performances. Corrélativement, les algorithmes des applications de traitement du signal et de l'image sont de plus en plus sophistiqués, comme le souligne abondamment l'évolution de standards vidéo MPEGxx et H26x, ou de standards 3G et 4G dans les communications. La mise en œuvre de ces applications sur un système embarqué parallèle devient complexe et les approches de prototypage rapide et de co-conception matérielle et logicielle sont alors souvent utilisées pour en faciliter le travail.

L'ordonnancement de tâches est une étape importante du prototypage rapide. L'objectif est alors de minimiser le temps d'exécution de l'application, en distribuant de manière rapide mais efficace, les tâches de calcul qui décrivent l'application sur les divers composants du système parallèle ciblé. La théorie des ordonnancements n'est pas nouvelle, le problème est reconnu NP-complet et les techniques sont multiples. Mais comme le poids des communications croît dans les applications les plus récentes, dans la compression vidéo comme dans les communications, l'intérêt d'améliorer ces techniques peut s'avérer pleinement justifié. Plusieurs techniques tiennent compte des poids des communications entre les tâches : certaines utilisent notamment un système de communication idéal [1,2,3,4], où des communications concurrentes sont exécutables simultanément. D'autres s'appuient sur un système plus réaliste [5,6,7], qui prend en compte la contention sur les communications.

Nous présentons ici, dans cet article, une heuristique d'ordonnancement qui tient compte de cette contention. Cette heuristique statique intègre de nouvelles règles de priorité basées sur les communications, qui impactent l'ordre des tâches à ordonnancer. Elle effectue aussi un ordonnancement partiel de la tâche successeur critique à la tâche courante à ordonnancer, afin d'affiner le choix du processeur à allouer à la tâche courante. En final, si ces techniques complexifient modérément l'heuristique d'ordonnancement, leur combinaison impacte toutefois l'ordonnancement de tâches et tend alors à accélérer l'exécution de l'application.

Ainsi, la section 2 suivante présente initialement les modèles de graphe utilisés pour décrire l'application et le système parallèle ciblé ; elle expose succinctement ensuite les principes d'un ordonnancement sensibilisé à de la contention sur les communications. La section 3 décrit alors les nouvelles techniques implantées, règles de priorité entre tâches et impact de la tâche successeur critique. Les résultats expérimentaux sont finalement commentés dans la section 4.

2 Modèles et définitions

Un graphe acyclique orienté G modélise l'algorithme de l'application à implanter. Un graphe topologique TG modélise le système ciblé multiprocesseurs.

2.1 Modèles de graphe

Le graphe acyclique orienté G = (N, E, *w*, c) modélise l'algorithme à implanter. N est l'ensemble des tâches n_i

du graphe, chacune considérée insécable ; $w(n_i)$ est la charge de calcul de la tâche n_i. La Figure 1 ci-après présente ainsi un exemple de graphe acyclique de neuf tâches. E est l'ensemble des arcs orientés e_{ij} : l'arc e_{ij} modélise le transfert d'informations de la tâche n_i vers la tâche n_j, de charge de communication $c(e_{ij})$.

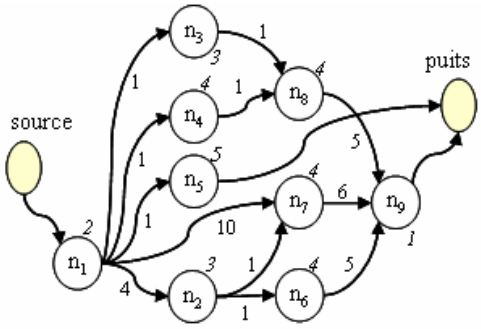

Figure 1 : exemple de graphe orienté G

Le graphe TG = (P, S, L, b) modélise la topologie de l'architecture ciblée multiprocesseurs. P est l'ensemble des processeurs p_k, éventuellement hétérogènes, qui exécutent séquentiellement les tâches. Ces processeurs sont interconnectés par un système de communication constitué de lien(s) bidirectionnel(s) l_i de l'ensemble L et de commutateur(s) s_j de l'ensemble S. La Figure 2 suivante présente ainsi trois exemples topologiques : a) trois processeurs p_k partageant un même bus l_0, b) six processeurs p_k pleinement interconnectés par le biais d'un commutateur s_1, c) deux clusters interconnectés de trois processeurs p_k chacun.

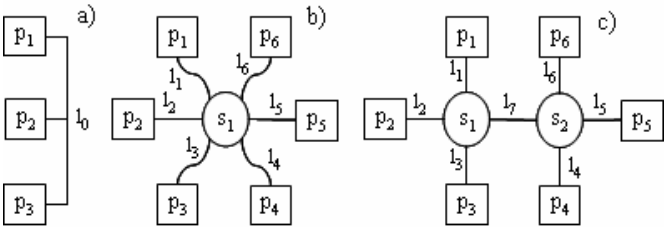

Figure 2 : exemples de graphe topologique TG

Tout commutateur s_j est modélisé idéalement : sa charge de calcul est nulle et il accepte le passage de communications simultanées, sans interaction. Ainsi, sur le graphe c), la communication de p_6 à p_5, via le routage $l_6 \rightarrow l_5$, et la communication de p_1 à p_4, via le routage $l_1 \rightarrow l_7 \rightarrow l_4$, sont simultanément implantables à travers s_2. La contention sur les communications est donc reportée sur la disponibilité des liens l_i, car en effet, l_i ne peut véhiculer qu'une unique communication dans un intervalle de temps déterminé. Dans cet article, les liens l_i sont considérés homogènes et de débit de données relatif $b(l_i)$ unitaire.

Le graphe TG ainsi défini permet de modéliser aussi bien la topologie de cartes multi-DSP, que la topologie de systèmes MPSoC prototypés dans des circuits FPGA par exemple.

2.2 Ordonnancement à contention sur les communications

L'ordonnancement d'un graphe G sur un graphe TG associe à chaque tâche n_i de N, un processeur p_k de P et une date de début d'exécution $t_d(n_i, p_k)$, en respectant les contraintes de précédence entre tâches. La date de fin d'exécution de n_i sur le processeur p_k est alors obtenue par $t_f(n_i, p_k) = t_d(n_i, p_k) + w(n_i, p_k)$. L'ordonnancement tient compte de la contention sur les communications en associant également, pour chaque communication e_{ij}, un routage complet $l_k \rightarrow \ldots \rightarrow l_m$ du processeur p_k au processeur p_m, et une date de début de communication $t_d(e_{ij}) = t_d(e_{ij}, l_k) = \ldots = t_d(e_{ij}, l_m)$. Une communication entre tâches ordonnancées sur un même processeur est négligée. La date de fin de communication entre deux processeurs est alors obtenue par $t_f(e_{ij}) = t_d(e_{ij}) + c(e_{ij})$. Un arc e_{ij} peut donc être ordonnancé, pour un routage défini et dans l'intervalle de temps [A, B] ($\in Q_+ \times Q_+$), si $\max(A, t_f(n_i, p_k)) + c(e_{ij}) \leq B$.

Puisque le système embarqué à cibler est défini avant l'ordonnancement, figeant ainsi le graphe TG, la table de routage des communications d'un processeur à un autre peut être élaborée statiquement à la compilation. Ainsi, l'ordonnancement de tâches s'avère pleinement à ressources contraintes. L'objectif est alors de minimiser la latence de l'ordonnancement, c'est-à-dire le temps global nécessaire pour exécuter l'ensemble des tâches de l'application, en fonction des ressources physiques allouées (processeurs et liens de communication).

3 Heuristique d'ordonnancement améliorée

La technique proposée pour le prototypage rapide s'appuie sur l'ossature d'un ordonnancement statique par liste, heuristique généralement privilégiée pour sa simplicité structurelle et sa rapidité dans la construction d'une solution, notamment dans l'utilisation au plus tôt des ressources matérielles. Cet ordonnancement peut être défini en trois procédures (cf. algorithme ci-après) : le tri ordonné des tâches du graphe G dans une liste statique, l'allocation d'un processeur à la tâche courante à ordonnancer, puis l'ordonnancement de ladite tâche.

Algorithme 1 : ordonnancement statique par liste

Entrées : G(N, E, w, c), TG(P, S, L, b)
Sortie : ordonnancement de G sur TG

---

1 *ListeTâches* ← **Tri_Tâches(** N **)**;
2 **pour** *chaque n ∈ ListeTâches* **faire**
3 p_{best} ← **Select_Processeur(** n, P **)**;
4 **Ordonne_Tâche(** n, p_{best} **)**;
5 **fait**

Cette section décrit les nouveautés apportées aux deux premières procédures dans l'heuristique proposée. Cette dernière développe alors P fois la complexité d'un ordonnancement par liste classique, soit une complexité en $O(P \times (N^2 + P \times E^2 \times O(\text{routage}))$.

3.1 Règles de priorité : impact sur le tri de tâches

L'ordre des tâches à ordonnancer impacte la latence de l'ordonnancement. [8] compare ainsi diverses règles de priorité sur les tâches et montre par expérimentation qu'un ordonnancement par liste basé sur un tri de tâches par ordre décroissant de leurs valeurs *bl*, surpasse tout

autre algorithme tenant compte de la contention. Une règle de priorité affecte à une tâche n_i, la valeur *bl* qui correspond à la charge globale du chemin critique aval établi entre la tâche n_i et la tâche puits du graphe G. De manière similaire, la règle détermine aussi la valeur *tl* qui correspond à la charge globale du chemin critique amont établi entre la tâche source du graphe G et la tâche n_i. La Figure 3 ci-dessous résume les cinq règles de priorité alors définies dans l'heuristique proposée : les tâches et les arcs en pointillé symbolisent la prise en compte des charges respectives *w()* et c(), dans le calcul itéré récursivement sur le graphe G, des valeurs *bl* et *tl* inhérentes à la règle pratiquée.

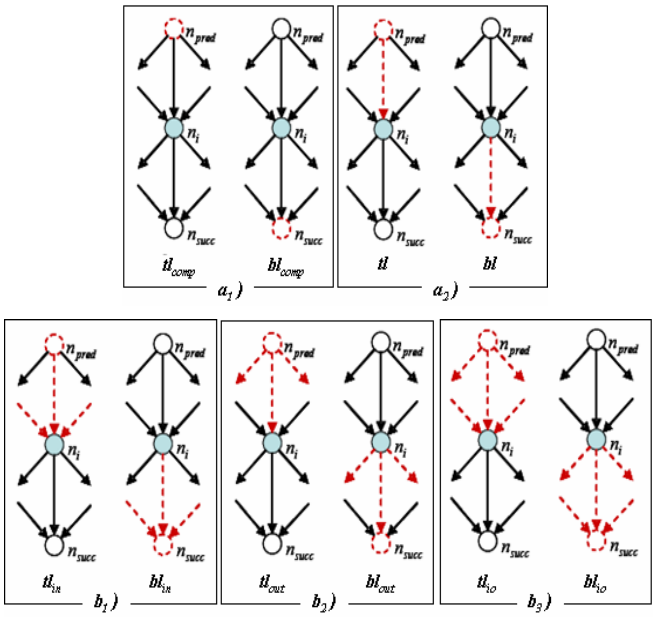

Figure 3 : règles de priorité récursives sur les tâches

Les règles a_k) sont notamment explicitées dans [7]. a_1) ne s'appuie que sur les charges de calcul *w()* des tâches qui forment les chemins critiques amont et aval définis précédemment ; a_2) y ajoute les charges $c(e_{ij})$ des arcs qui établissent les communications de ces chemins critiques. Les trois nouvelles règles b_k) tiennent compte de la contention sur les communications, en impactant les valeurs *bl* et *tl* de chaque tâche de G par les charges des communications entrantes « *in* » dans b_1), sortantes « *out* » dans b_2), entrantes et sortantes « *io* » dans b_3). Le tableau suivant présente ainsi les valeurs obtenues pour deux des règles appliquées aux dernières tâches du graphe G décrit en exemple par la Figure 1.

Tab 1 : règles appliquées au graphe G de la Figure 1

Tâches		n_4	n_5	n_6	n_7	n_8	n_9
a_2)	*tl*	3	3	10	12	8	22
	bl	15	5	10	11	10	1
b_2)	tl_{out}	19	19	24	24	24	34
	bl_{out}	15	5	10	11	10	1
$sc(n_i)$		n_8	ϕ	n_9	n_9	n_9	ϕ

L'heuristique proposée élabore les valeurs inhérentes à une règle de priorité donnée. Ensuite, sous vérification des contraintes de précédence, elle trie les tâches du graphe dans une liste statique, par ordre décroissant des valeurs *bl* (et par ordre croissant des valeurs *tl* pour les tâches de même priorité). Il est évident que l'application de règles différentes sur un même graphe peut amener à des listes ordonnées de tâches, distinctes.

3.2 Allocation du processeur à la tâche courante : technique du successeur critique

L'ordonnancement par liste alloue classiquement à la tâche courante n_i, le processeur p_k qui propose la date de fin $t_f(n_i, p_k)$ la plus tôt. Cette approche tend à optimiser localement l'ordonnancement, qui peut finalement se révéler parfois décevant globalement. Aussi, [4] propose d'affiner cette allocation à la tâche n_i, en tenant compte de l'ordonnancement de la tâche successeur critique $sc(n_i)$ de n_i, dans le cas d'une sélection dynamique de n_i et d'un nombre de processeurs non limité et pleinement interconnectés. L'heuristique proposée dans cet article intègre ce concept de successeur critique, mais adapté toutefois à une sélection statique des tâches et surtout à un graphe topologique figé, c'est-à-dire pour un nombre contraint de processeurs et dans le cas de la contention sur les communications. $sc(n_i)$ est ainsi redéfinie en une tâche successeur de n_i qui émerge prioritairement de la liste des tâches triées statiquement. L'ordonnancement de $sc(n_i)$ impacte le choix du processeur à allouer à la tâche courante n_i, même si cet ordonnancement peut être partiel ou faussé, puisque certaines tâches prédécesseurs de $sc(n_i)$ et les communications associées, peuvent ne pas être encore ordonnancées. Finalement, le choix de la tâche successeur critique à une tâche courante dépend du tri ordonné des tâches et donc de la règle de priorité qui a été appliquée : les deux techniques sont ainsi corrélées pour impacter la latence de l'ordonnancement.

4 Résultats expérimentaux

L'application des cinq règles de priorité sur le graphe G de la Figure 1 aboutit à trois listes distinctes de tâches triées. Le tableau suivant les présente : (n_i, n_j) symbolise des tâches de priorité identique (mêmes *bl* et mêmes *tl*), ordonnées alors aléatoirement.

Tab 2 : ordres des tâches du graphe G de la Figure 1

Règles	Liste ordonnée de tâches
a_1)	n_1, n_4, (n_3, n_2), n_8, (n_7, n_6), n_5, n_9
a_2) - b_1) - b_2)	n_1, n_2, n_4, n_3, n_7, n_6, n_8, n_5, n_9
b_3)	n_1, n_2, n_4, n_3, n_7, n_8, n_6, n_5, n_9

La Figure 4 page suivante compare l'ordonnancement par liste classique à l'ordonnancement de la liste triée suivant la règle de la Figure 3-b_3), qui tient compte des communications entrantes et sortantes dans les chemins critiques (cf. section 3.1). Le système ciblé est illustré par la Figure 2-a), où chaque processeur p_k dispose d'un communicateur, typiquement un DMA, permettant alors l'exécution simultanée d'une tâche de calcul et d'une communication sur le bus. Les ordonnancements sont équivalents pour les cinq tâches ordonnées, de n_1 à n_7. L'approche classique ordonnance ensuite au plus tôt n_8 sur p_3, puis n_6 sur p_2 ; l'approche proposée ordonnance également n_8, mais la prise en compte de son successeur critique (n_9, cf. tableau Tab 1), induit la tâche n_8 vers le

processeur p_1, au lieu de p_2 ou p_3 qui intuitivement optimiserait l'ordonnancement localement. Au final, les latences diffèrent de quatre pour le graphe G étudié, au détriment de l'heuristique classique.

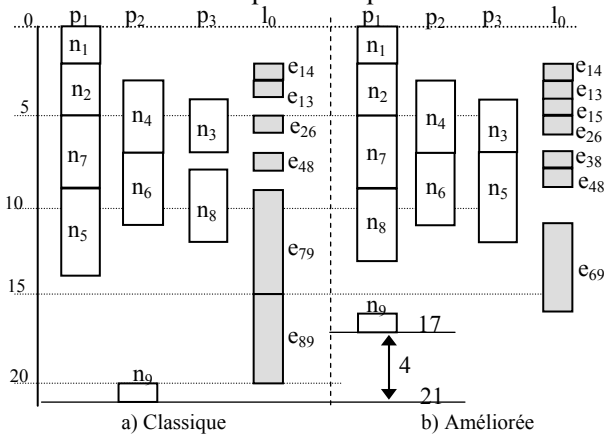

Figure 4 : exemples d'ordonnancement

La règle de priorité adéquate varie évidemment d'un graphe G à un autre et l'obtention d'un critère de choix performant assujetti à la structure du graphe ou à sa complexité n'est pas aisée. Toutefois, l'exécution de l'heuristique, pour une règle précise, sur un graphe de 500 nœuds et un système de 16 processeurs, prend quelques dizaines de secondes seulement sur un PC de bureau classique. Aussi, l'heuristique proposée applique en fait les cinq règles de priorité sur le graphe, génère au plus cinq listes distinctes de tâches triées, puis effectue ensuite autant d'ordonnancements : le meilleur résultat (latence) est finalement conservé. Le temps d'exécution de l'ensemble prend environ quelques minutes pour les graphes expérimentés, durée jugée admissible dans le cadre du prototypage rapide.

Pour vérifier statistiquement la stabilité, l'heuristique applique l'ensemble des cinq règles et la technique *sc()* sur des groupes de graphes générés aléatoirement. Le facteur d'accélération, rapport de la latence obtenue classiquement sur la latence obtenue par l'heuristique proposée, est le critère de comparaison retenu. La figure ci-dessous présente alors des résultats qui soulignent la stabilité de l'approche et montre son impact sur la latence de l'ordonnancement, dans le cas notamment de situations à forte communication (CCR=10).

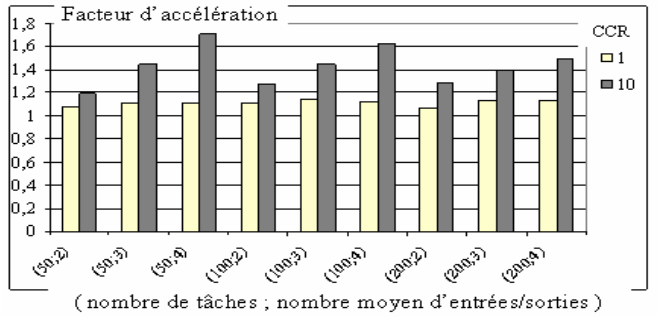

Figure 5 : ordonnancements sur graphes aléatoires

Ces résultats sont obtenus pour des groupes de 1000 graphes générés par le biais du générateur aléatoire de graphe SDF^3 décrit dans [9], pour des nombres moyens identiques d'entrées et sorties par tâche, pour des charges de calcul variant aléatoirement entre 10^2 et 10^3 et des charges de communication variant entre 10^2xCCR et 10^3xCCR. CCR est le rapport entre la moyenne des charges de communication et la moyenne des charges de calcul : il prend typiquement les valeurs 0.1, 1 ou 10, pour représenter des situations à faible, à moyenne ou à forte communication.

5 Conclusion

Deux techniques tenant compte de la contention sur les communications sont présentées dans cet article et sont intégrées dans une heuristique d'ordonnancement par liste statique appliquée pour ordonnancer les tâches d'une application de traitement d'images sur un système multiprocesseurs. Ainsi, les nouvelles règles de priorité impactent le tri statique des tâches à ordonnancer et la technique du successeur critique affine l'allocation d'un processeur à une tâche courante. Les expérimentations montrent la stabilité de l'approche et son efficacité dans la minimisation de la latence de l'application, pour des situations à moyenne et à forte communication. Comme les communications s'intensifient dans les applications récentes, notamment en compression vidéo et dans les communications, l'heuristique proposée semble alors appropriée pour prototyper rapidement ces applications sur des systèmes embarqués parallèles.

Références

[1] J-J.Hwang, Y-C.Chow, « Scheduling Precedence Graphs in Systems with Interprocessor Communication Times », *SIAM J.Comput.*, vol.18, n°2, pp.244-257, 1989.

[2] M-Y.Wu, D.Gajski, « Hypertool : a Programming Aid for Message-Passing Systems », *IEEE Trans. on Parallel and Distributed Systems*, vol.1, n°3, pp.330-343, 1990.

[3] T.Yang, A.Gerasoulis, « DSC : Scheduling Parallel Tasks on an Unbounded Number of Processors », *IEEE Trans. on Parallel and Distributed Systems*, vol.5, n°9, pp.951-967, Sept. 1994.

[4] YK.Kwok, I.Ahmad, « Dynamic Critical-Path Scheduling : an Effective Technique for Allocating Task Graphs on Multiprocessors », *IEEE Trans. on Parallel and Distributed Systems*, vol.7, n°5, pp.506-521, May 1996.

[5] G.Sih, E.Lee, « A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures », *IEEE Trans. on Parallel and Distributed Systems*, vol.4, pp.175-187, Feb. 1993.

[6] T.Grandpierre, C.Lavaenne, Y.Sorel, « Optimized Rapid Prototyping for Real-Time Embedded Heterogeneous Multiprocessors », in *Proceedings of 7th Int. Workshop on Hardware/Software Co-Design*, *CODES'99*, Rome, Italy, May 1999.

[7] O.Sinnen, L.Sousa, « Communication Contention in Task Scheduling », *IEEE Trans. on Parallel and Distributed Systems*, vol.16, n°6, pp.503-515, June 2005.

[8] O.Sinnen, L.Sousa, « List Scheduling : Extension for Contention Awareness and Evaluation of Node Priorities for Heterogeneous Cluster Architectures", *Parallel Computing*, vol.30, n°1, pp.81-101, Jan. 2004.

[9] S.Stujik, M.Geilen, T.Basten, « SDF^3 : SDF for Free", in *Proceedings of 6th Int. Conference of Application of Concurrency to System Design*, *ACDS'06*, pp.276-278, USA, June 2006.